UNLIMITED

(2)

RSRE
MEMORANDUM No. 4515

ROYAL SIGNALS & RADAR ESTABLISHMENT

USING ELLA FOR HIGH LEVEL DESIGN: MODELLING THE MOTOROLA 6800 MICROPROCESSOR

Author: A R Huggett

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 4515

UNLIMITED

CONDITIONS OF RELEASE

0110616 305232

********** DRIC U

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations

Royal Signals and Radar Establishment

Memorandum 4515

Title: Using ELLA for High Level Design:

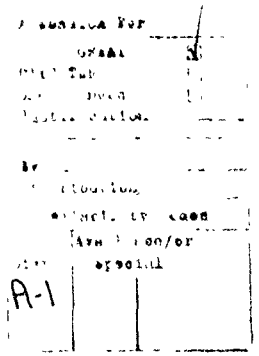
Modelling the Motorola 6800 Microprocessor.

Author: A R Huggett (Student Engineer)

Date: 5 September 1991

Summary

This document starts by providing a brief overview of the ELLA language. The majority of the document describes the design and development of an abstract high level ELLA model of a Motorola 6800 Microprocessor, including all of the 197 codes which make up the 6800 instruction set.



Disclaimer:

The author gives no guarantee that the model herein described is a fully accurate and error free description of the Motorola 6800

The views represented within this report are those of the author alone and do not necessarily represent the views of the Ministry of Defence nor the Defence Research Agency

Notices:

Throughout the text the symbol & is used to indicate a hexadecimal number. However, in the ELLA language this is replaced by 16r, thus address &e000 becomes ad/16re000 in ELLA

Variables and Types which occur within the 6800 model are indicated by **bold** type.

Words in the glossary are indicated in *italics*.

References to the Bibliography are indicated thus [1]

Contents:

The ELLA Language

ELLA is a Textual Language for describing digital hardware designs. The language is designed so that ELLA circuit descriptions are always realisable in hardware.

The components of a circuit are described in ELLA by functions. A function definition declares the number and types of inputs and outputs. The function body is composed of a number of constructs, which describe how the inputs map to the outputs

Before any functions may be written in ELLA it is necessary to define the variable types which they will use This is achieved by means of the TYPE statement, e.g.

```
TYPE count = NEW co/(0..12)
TYPE addrmode = NEW (immed|direct|index|extnd|implied|relative)
```

The first statement defines TYPE count to be an integer in the range 0..12. The co/ tag is necessary to distinguish that type from any other integer type The second statement defines TYPE addrmode as a six valued enumerated type, with possible values of immed, direct, index, extnd, implied and relative

ELLA supports a wide variety of high level constructs such as ARITH which performs integer arithmetic, DELAY which delays information between its input and output and provides a means of feedback from one clock cycle to the next

The language is strongly typed in order to minimise the chances of the user introducing a design fault. It has a hierarchial functional structure similar to PASCAL.

The main difference between ELLA and conventional programming languages such as C, PASCAL etc is that it is a parallel rather than a sequential language This is necessary to describe logic circuits However since it is a parallel language, to try and express

```
LET a = da/5
LET a = da/4
```

is illegal since the compiler tries to implement both of these statements in parallel A variable defined by LET is therefore static once assigned within a timestep its value cannot be changed within that timestep although it can be used as an input to other functions The value of a static variable can be changed on a different timestep however

For some applications, however, it is more convenient to program sequentially, particularly for complex high level descriptions The SEQ construct allows sequential programming, with two types of dynamic variables, which must be defined before they can be used, but once defined may be assigned to many times within the sequence The first of these is VAR VARs are initialised to a predefined value every timestep. The other type is PVAR (or STATE VAR) These are initialised at time 0 but retain the latest assigned value from one timestep to the next until they are reassigned They are very useful for making registers. Sequential constructs can be directly transformed into parallel ones using a set of assembler tools

Care must be taken to ensure that the same instantiation of the function is being used if the old value of the PVAR is required In practice this means that functions which contain constructs such as VAR, PVAR, or RAM and DELAY functions become wrapped up inside parallel functions which allow explicit instantiation by using the MAKE and JOIN constructs

For a complete description of the ELLA Language refer to the ELLA Manual[1]

The Motorola 6800 Model: Specification

My model of the Motorola 6800 was developed from a very limited amount of information I had the instruction set, the assembler mnemonics, number of bytes, number of clock cycles required and effect of the function[2] I did not know anything of the order in which internal subfunctions are carried out within the microprocessor My knowledge of and information on the *condition codes register* was also less than ideal

Since I had no information about the internal operations of the 6800 I decided that my model should fulfil three criteria·

i) It should perform the correct instruction for the appropriate instruction code.

ii) It should take the correct number of clock cycles to complete each instruction.

iii) All 197 valid *op-codes* should be implemented.

In addition I wished the model to have what I hoped was a realistic internal architecture

The Motorola 6800 Model: Variable Type Selection

Nine different variable types are used within the ELLA description (see Appendix A)
The reason for this is as follows

The Motorola 6800 is an eight-bit microprocessor Therefore all data coming into and out of the microprocessor is eight bits wide and TYPE **data** provides this The choice of TYPE **data** to be positive in the range 0..255 rather than signed -128 to 127 was made for two reasons.

i) The mathematics of instructions such as **rola** become easier to implement
ii) Logic synthesis is often easier if there are no negative numbers

Since the address space of the 6800 is 0 65535 (&ffff) it was clear that a sixteen bit type. TYPE **address** was needed This time it was obvious that the numbers should be positive

A four bit type, TYPE **halfbyte** proved useful within the instruction decoding functions, because the more significant four bits of an op-code always refers to the addressing mode and the less significant to the instruction

TYPE **flag** is a two state boolean which is useful in CASE clauses

TYPE **bitint** is the integer equivalent of TYPE **flag**, it is used in some ARITH statements as the carry bit

TYPE **addrmode** and TYPE **mnemonic** are enumerated types which represent the addressing mode and mnemonic of an op-code in a user-friendly way

TYPE **count** is used in the microprocessor's internal structure to determine how far through the instruction it is Since the longest instruction takes 12 cycles the maximum value of TYPE **count** is 12

TYPE **result** is the 9-bit answer of the main ARITH functions within FN ALU It is necessary because the 9th bit is important for setting the cc register

The Motorola 6800 Model: Initial Design

The Motorola 6800 is a simple 8-bit *Von-Neumann microprocessor* which has two 8-bit *accumulators*, **a** and **b**, an index register **x**, a *stack* pointer **sp** and a *condition codes register* **cc**. It also has a program counter **pc**. In addition I found it useful to create an address buffer **addrbuf** which I now know is not present on the real chip.

I originally had to decide whether it would be better to build a model which interpreted 6800 *assembler code* or hexadecimal op-codes. I decided that using the memory to store mnemonics as a separate type from data was too far away from reality, and too difficult to implement. Therefore I opted for a memory interface which just used 8-bit data. However, in order to make the design easier to program and to understand I decided that instructions should be stored within the microprocessor as an assembler *mnemonic* plus an *addressing mode*. Thus FN MNEMONIC and FN ADMODE are used to extract this information when the incoming data word is an instruction (see Appendix A).

Studying the data I found that with one exception (which I erroneously took to be a printing error), the addressing mode affected the number of cycles required to perform an instruction as follows.

Implied	+0
Immediate	+1
Direct	+2
Relative/Extended	+3
Indexed	+4

I deduced that the microprocessor must have some way of knowing how far through an instruction it has proceeded since all instructions take between 2 and 12 timesteps. I therefore created the model with an internal counter (**intercount**) which operates as follows

If **intercount** is zero then the incoming byte on the databus is treated as an instruction, and the functions ADMODE and MNEMONIC are used to change the instruction byte into an addressing mode and a mnemonic From these the function HOWLONG calculates how many clock cycles the mnemonic should take, plus an offset for the addressing mode Intercount is then set to the value returned from FN HOWLONG In order to facilitate future logic synthesis I have avoided the use of negative numbers Thus FN COMPARECOUNT always returns a positive value or zero if the second input is greater than or equal to the first FN COMPARECOUNT is then called to see how far into the addressing mode cycle the microprocessor has proceeded, by comparing intercount with the number of cycles required to perform the instruction if there were no addressing steps, and the appropriate addressing step is carried out If the value returned from FN COMPARECOUNT is zero then the comparison is repeated with the inputs reversed so as to provide a result which might be positive or zero, and the appropriate instruction step carried out

In order to test the microprocessor it is necessary to connect it to a memory, so I created FN MAKERAM and FN HARNESS which provide this plus a reset line

The first instructions to be implemented were **adda, ldaa, suba, beq, jmp** and **staa** (see Appendix C) These were tested by means of two test programs, TESTPROG1 and TESTPROG2 (see appendix B)

The Motorola 6800 Model: Development

Having got the memory and the first six instructions working to my satisfaction I began increasing the instruction set After some time it became apparent that the length of the main program loop was getting too big to be easily understood, and furthermore, was beginning to get unrealistic with 16 bit adders creeping in that do not exist in the real 6800 I therefore decided to change the way in which arithmetic and logical functions were being implemented. I created FN ALU (Arithmetic and Logical Unit) which had basic functions defined. These were FN ADDBYTE, FN ORBYTE, FN NOTBYTE, FN ANDBYTE and FN SRBYTE. (ADDBYTE performs binary add, SRBYTE binary shift right with wraparound, ORBYTE binary OR etc.) There were also several functions for such things as truncating 9-bit results (FN CHOPDATA) and condition codes register handling. Later I added a sixth main instruction, FN DAABYTE which performs a decimal adjust on accumulator a.

In order to implement 16-bit operations such as **inx** (see Appendix C) I found it necessary to pass information as to which part of an instruction was to be executed by the ALU I achieved this without increasing the number of inputs by utilising the two most significant bits of the condition codes register, which the data sheet defines as both being true at all times, and therefore redundant

The text was tidied by using functions such as FN BRANCH to wrap up functions of similar type The functions which take up a lot of space are those such as **jsr** and **swi** (see Appendix C) which take many timesteps and are fairly unique. A lot of expansion to FN ALU was also required

Finally I implemented the *interrupt request*, *non-maskable interrupt* and halt lines into the microprocessor These are separate level sensitive lines which are inverted, i.e a logic false causes an *interrupt* to occur or the microprocessor to cease all operation accordingly. The handling of these interrupts is virtually identical to the **swi** instruction. I therefore decided that by far the easiest way to implement the interrupt was to have two dummy mnemonics, **irq** and **nmi**.

I also memory mapped three inputs into the computer and three outputs from it Locations &f000 to &f002 if read from by the microprocessor return the three data inputs to FN HARNESS Similarly locations &e000 to &e002 if written to change the output of FN HARNESS This arrangement can be thought of as being equivalent to a set of three input registers which can be loaded by using a keypad or other means, and six hexadecimal output digits.

With the microprocessor now theoretically fully functional I wrote an initialisation program which clears all the registers to definite values, and a monitor program which is called by a non_maskable interrupt

The Motorola 6800 Model: Testing

The development of the model was a repetitive cycle of adding more instructions, removing compilation errors and then testing by means of test programs (Appendix B). These were called into the simulator as ELLA instruction (*.eli) files which initialised the ram to the values required The majority of programs perform the same function: they calculate the first ten numbers in the Fibonacci series which are calculated by adding the previous two terms in the series to get the current one. Assembler mnemonics are provided as comments in the programs to assist the reader.

Due to the large number of instructions (197 valid op-codes, 2 other interrupt functions, a reset function and a halt line), it was not possible to test every instruction in the time available. however I believe that the instructions contained within the test programs represent a good cross section The complex instructions such as **jmp**, **swi**, **irq**, **nmi**, **rti**, **rts**, **jsr** and **bsr** have all been tested, however some simple mathematical functions such as **addb** are assumed because **adda** works and **addb** is a direct copy (see Appendix A)

The Motorola 6800 Model: Conclusions

My model of a Motorola 6800 microprocessor fulfils the 3 specification requirements set out above However there may be some errors in the condition codes register; these could definitely be found and fixed with some more testing time.

The internal architecture is quite unlike the real 6800 The Motorola data sheet[3] became available very late in the development stage when I was implementing interrupts. With a few days work, however, I believe that the program could be modified to give it a much better resemblance to the 6800 internally

Given the same project again I would implement the timing of addressing mode operations differently, and would try and mimic the two phase clock required by the 6800 (my current model has only a single phase). There would be no addrbuf register and all internal registers would be 8-bits wide With the use of the Motorola data sheet I could ensure that every register contained the right information on every cycle (the current model is only correct at the end of each instruction).

The development of my model was largely based upon a very limited amount of information This lack of information lead to a number of differences between it and the real 6800. However I believe that this model demonstrates that a fully functional ELLA description can be obtained with limited information and minimal difficulty which, when viewed from outside the microprocessor/memory system behaves identically to the real thing

Appendix A: The ELLA Description

The ELLA description below is arranged in a compilable order i e declare before use Thus the main microprocessor function FN UP6800 is towards the end of the listing

```
#Variable Type Definitions#
TYPE data = NEW da/(16r00 16rff),
    address = NEW ad/(16r0000..16rffff),
    result = NEW re/(16r000..16r1ff),
    bitint = NEW bi/(0 1),
    flag = NEW (h|l),
    count = NEW co/(0..12),
    addrmode = NEW (immed|direct|index|extnd|implied|relative),
    halfbyte = NEW hb/(16r0 16rf),
    mnemonic = NEW
(adca|adda|anda|bita|clra|cmpa|coma|deca|eora|inca|oraa|psha|pshb|pulb|pula|rola
|rora|asla|asra|lsra|nega|ldaa|suba|staa|sbca|tsta|cpx|dex|des|inx|ins|ldx|lds|
stx|sts|txs|tsx|bra|bcc|bcs|beq|bge|bgt|bhi|ble|bls|blt|bmi|bne|bvc|bvs|bpl|bsr|
jmp|jsr|nop|addb|adcb|andb|bitb|clrb|cmpb|comb|decb|eorb|ldab|incb|negb|orab|
rolb|rorb|aslb|asrb|lsrb|sbcb|subb|tstb|stab|tba|tab|aba|cba|daa|sba|clr|dec|inc
|com|neg|rol|ror|asl|asr|lsr|tst|rti|rts|swi|wai|clc|cli|clv|sec
|sei|sev|tap|tpa|irq|nmi)

#Bit Tester#

FN TRUEBIT = (data input control) -> flag
ARITH IF (input IAND control) = control
      THEN 1 #h#
      ELSE 2 #l#
    FI
```

```
#Arithmetic and Logical Unit#

FN ALU = (mnemonic: inst, data: input1 input2 ccreg) -> [2]data:

BEGIN SEQ

  FN ADDBYTE  = (data ip1 ip2, bitint.ip3) -> result· ARITH (ip1+ip2+ip3),

  FN ANDBYTE  = (data·ip1 ip2)          -> result· ARITH ip1 IAND ip2,

  FN ORBYTE   = (data ip1 ip2)          -> result  ARITH ip1 IOR  ip2,

  FN NOTBYTE  = (data.ip1)              -> result  ARITH (INOT ip1) IAND
                                              16r1ff;

  FN CHOPDATA = (result ip1)            -> data  · ARITH ip1 IAND 16rff;

  FN SRBYTE   = (result ip1)            -> result   ARITH (ip1 SR 1)
                                           +256*(ip1 IAND 1);
  FN DAABYTE = (data ip1 ccreg) -> result.
    ARITH ip1+IF ((ip1 IAND 16rf)>9)
             OR ((ccreg IAND 16r20) = 16r20)
             THEN IF (ip1 IAND 16rf0) = 16r90 THEN 16r66 ELSE 6 FI
             ELSE IF (ip1 IAND 16rf0) > 16r90 THEN 16r60 ELSE 0 FI
          FI,

  FN HALFCARRY= (data ip1 ip2 ccreg, bitint ip3) -> data
    ARITH (ccreg IAND 16rdf)+
    IF ((ip1 IAND 16r0f)+(ip2 IAND 16r0f)+ip3)>16r0f THEN 16r20 ELSE 0 FI,

  FN ACARRY = (result ip1, data ccreg) -> data
    ARITH (ccreg IAND 16rfe)+IF ip1>255 THEN 1 ELSE 0 FI,

  FN SCARRY = (result ip1, data ccreg) -> data
    ARITH (ccreg IAND 16rfe)+IF ip1>255 THEN 0 ELSE 1 FI,



  FN DCARRY = (result ip1, data ccreg)  > data
    ARITH ccreg IOR IF ip1>255 THEN 16r01 ELSE 0 FI,

  FN OVERFLOW1 =(data ip1 ip2 output ccreg) -> data
    ARITH (ccreg IAND 16rfd)+IF ((ip1 IAND 16r80)=(ip2 IAND 16r80))
                        AND ((ip1 IAND 16r80) /= (output IAND 16r80))
                        THEN 16r02
                        ELSE 0
                       FI,
```

```
FN OVERFLOW2 = (data: ip1 ip2 output ccreg) -> data·
  ARITH (ccreg LAND 16rfd) + IF ((ip1 LAND 16r80) = (((INOT ip2) + 1) LAND 16r80))
                    AND ((ip1 LAND 16r80) /= (output LAND 16r80))
                    THEN 16r02
                    ELSE 0
                  FI,


FN OVERFLOW3 = (data ccreg) -> data: ARITH (ccreg LAND 16rfd),


FN OVERFLOW4 = (result:ip1,data ccreg) -> data·
   ARITH (ccreg LAND 16rfd) +
     IF (((ip1 LAND 16r100) SR 1) LAND (ip1 LAND 16r80)) =
       (((ip1 LAND 16r100) SR 1) IOR (ip1 LAND 16r80)) THEN 0 ELSE 2 FI;


FN OVERFLOW5 = (data·ip1 ccreg) -> data
   ARITH (ccreg LAND 16rfd) + IF ip1 = 16r80 THEN 2 ELSE 0 FI,


FN ZERO = (data· ip1 ccreg) -> data
   ARITH (ccreg LAND 16rfb) + IF ip1 = 0 THEN 4 ELSE 0 FI,


FN NEG = (data ip1 ccreg) -> data
   ARITH (ccreg LAND 16rf7) + IF ip1 > 127 THEN 8 ELSE 0 FI;

FN CCCOUNT = (data ccreg) -> data
   ARITH (ccreg + 16r40) LAND 16rff,

FN CCRESET = (data ccreg) -> data
   ARITH ccreg IOR 16rc0,

VAR answer = re'0,
   output = da'0,
   newcc = ccreg,
```

```
CASE inst
  OF adda|addb|aba (answer := ADDBYTE(input1,input2, bi/0);
                                    output:=CHOPDATA answer),
    adca|adcb (answer = ADDBYTE(input1, input2, CASE TRUEBIT(ccreg,da/1)
                                    OF h bi/1
                                    ELSE bi/0
                                  ESAC),output = CHOPDATA answer),
    anda|andb: (answer := ANDBYTE (input1, input2);output:= CHOPDATA answer),
    bita|bitb: (answer := ANDBYTE (input1, input2);output:= input1),
    clra|clrb|clr (answer := re/0;output := da/0),
    cmpa|cmpb|cpx|cba (   wer = ADDBYTE (input1, CHOPDATA NOTBYTE input2, bi/1);
                                    output:=CHOPDATA answer),
    coma|comb|com: (answer := NOTBYTE input1, output.= CHOPDATA answer),
    daa: (answer := DAABYTE (input1,ccreg), output := CHOPDATA answer),
    nega|negb|neg (answer := ADDBYTE (CHOPDATA NOTBYTE input1,da/0,bi/1);
                                    output:= CHOPDATA answer),
    deca|decb|dec: (answer := ADDBYTE (input1, da/255, bi/0);output.=CHOPDATA answer),
    eora|eorb (answer := ANDBYTE (CHOPDATA ORBYTE (input1,input2),
                  CHOPDATA NOTBYTE CHOPDATA ANDBYTE (input1,input2)),
                  output := CHOPDATA answer),
    inca|incb|inc (answer = ADDBYTE (input1, da/1, bi/0);output := CHOPDATA answer),
    ldaa|staa|ldab|stab|ldx|stx|lds|sts|tab|tba.
                  (answer =ADDBYTE(da/0,input2,bi/0);output := input2),
    oraa|orab (answer = ORBYTE (input1, input2),output = CHOPDATA answer),
    rola|rolb|rol (answer = ADDBYTE (input1, input1,CASE TRUEBIT(ccreg,da/1)
                                    OF h.bi/1
                                    ELSE bi/0
                                  ESAC),output = CHOPDATA answer),
    rora|rorb|ror (answer = SRBYTE ADDBYTE (input1, CASE TRUEBIT(ccreg,da/1)
                                    OF h  da/255
                                    ELSE da/0
                                  ESAC,
                                  CASE TRUEBIT(ccreg,da/1)
                                    OF h  bi/1
                                    ELSE bi/0
                                  ESAC),output = CHOPDATA answer),
    asla|aslb|asl (answer = ADDBYTE (input1,input1, bi/0),output = CHOPDATA answer),
    asra|asrb|asr (answer = SRBYTE ADDBYTE (input1, CASE TRUEBIT(ccreg,da/1)
                                    OF h  da/255
                                    ELSE da/0
                                  ESAC,
                                  CASE TRUEBIT(ccreg,da/1)
                                    OF h  bi/1
                                    ELSE bi/0
                                  ESAC),output = CHOPDATA answer),
    lsra|lsrb|lsr (answer = SRBYTE ADDBYTE (input1, da/0, bi/0),
                                    output = CHOPDATA answer),
    suba|subb|sba (answer =ADDBYTE (input1, CHOPDATA NOTBYTE input2, bi/1),
                                    output = CHOPDATA answer),
```

```
    sbca¦sbcb. (answer := ADDBYTE (input1, CHOPDATA NOTBYTE input2,
                                   CASE TRUEBIT(ccreg,da/1)
                                     OF h:bi/0
                                     ELSE bi/1
                                   ESAC);output:= CHOPDATA answer),
    tsta¦tstb¦tst· (answer := ADDBYTE (input1, da/0, bi/0),
                                     output·= CHOPDATA answer),
    dex¦des  (answer :=CASE TRUEBIT (ccreg,da/16r40)
                      OF h: ADDBYTE (input1, da/255, bi/0),
                        l: CASE input2
                             OF da/16rff ADDBYTE(input1,da/255,bi/0)
                             ELSE ADDBYTE (input1, da/0, bi/0)
                           ESAC
                  ESAC; output ·= CHOPDATA answer),
    inx¦ins  (answer ·=CASE TRUEBIT (ccreg,da/16rc0)
                      OF h. ADDBYTE (input1, da/1, bi/0),
                        1 CASE input2
                             OF da/16r00  ADDBYTE(input1,da/1,bi/0)
                                ELSE    ADDBYTE(input1,da/0,bi/0)
                           ESAC
                  ESAC, output  = CHOPDATA answer)
ESAC,

newcc ·= CASE inst
         OF adda¦addb¦aba  HALFCARRY (input1,input2, newcc, bi/0),
           adca'adcb. HALFCARRY (input1,input2, newcc, CASE TRUEBIT(ccreg,da/1)
                                                         OF h  bi/1
                                                         ELSE bi/0
                                                       ESAC)
         ELSE newcc
       ESAC,

newcc  = CASE inst
          OF adda¦addb¦aba¦adca¦adcb¦clra'clrb¦coma¦comb¦nega¦
            negb¦rola¦rolb¦rora¦rorb¦asla¦aslb¦asra¦asrb¦lsra¦lsrb
            ¦tsta¦tstb¦clr¦com¦neg¦rol¦ror¦asl¦asr¦lsr¦tst
                 ACARRY (answer,newcc),
            daa  DCARRY (answer,newcc),
            cmpa¦cmpb¦cba¦subb¦sbca¦sbcb¦suba¦sba  SCARRY (answer,newcc)
          ELSE newcc
       ESAC,
```

```
newcc := CASE inst
         OF adda|adca|adcb|addb|aba|daa·OVERFLOW1 (input1,input2,output,newcc),
           inca|incb|inc  OVERFLOW1 (input1,da/1,output,newcc),
           suba|subb|sbca|sbcb|cmpa|cmpb|cba|sba·OVERFLOW2 (input1,input2,
                                        CHOPDATA answer,newcc),
           deca|decb|dec  OVERFLOW2 (input1,da/1,output,newcc),
           anda|andb|bita|bitb|clra|clrb|coma|comb|eora|eorb|ldaa|ldab|tab
           |tba|oraa|orab|staa|stab|tsta|tstb|clr|com|tst
                                        OVERFLOW3(newcc),
           rola|rolb|rora|rorb|asla|aslb|asra|asrb|lsra|lsrb.OVERFLOW4 (answer,newcc),
           nega|negb|rol|ror|asl|asr|lsr|neg: OVERFLOW5 (output,newcc),
           cpx: CASE TRUEBIT(ccreg,da/16r40)
                  OF 1 OVERFLOW2 (input1,input2,CHOPDATA answer,newcc)
                  ELSE newcc
                ESAC
           ELSE newcc
       ESAC;


newcc := CASE inst
         OF cpx|dex|inx|ldx|lds|stx|sts
              CASE TRUEBIT (ccreg,da/16rc0)
                 OF h. ZERO (output, newcc),
                   l· CASE TRUEBIT (ccreg,da/16r04)
                        OF h. ZERO (output,newcc)
                        ELSE newcc
                      ESAC
              ESAC,
           des'ins  newcc
         ELSE ZERO (output,newcc)
       ESAC,


newcc . = CASE inst
         OF dex|des|ins|inx  newcc,
           cpx|ldx|lds|stx|sts  CASE TRUEBIT (ccreg,da/1or40)
                                  OF h  NEG (output,newcc)
                                  ELSE newcc
                                ESAC
         ELSE NEG (output,newcc)
       ESAC,

newcc  = CASE inst
        OF clc  CHOPDATA ANDBYTE (ccreg, da/16rfe),
          cli  CHOPDATA ANDBYTE (ccreg, da/16ref),
          clv  CHOPDATA ANDBYTE (ccreg, da/16rfd),
          sec· CHOPDATA ORBYTE  (ccreg, da/16r01),
          sei  CHOPDATA ORB\"TE  (ccreg, da/16r10),
          sev. CHOPDATA ORB r .E  (ccreg, da/16r02),
          tap  CHOPDATA ORBYTE  (input1, da/16rc0)
         ELSE newcc
       ESAC,
```

```
newcc  = CASE inst
         OF cpx¦dex¦des¦inx¦ins¦ldx¦lds¦stx¦sts
           CASE TRUEBIT (ccreg,da/16rc0)
              OF h. CCCOUNT(newcc),
                 l: CCRESET(newcc)
           ESAC
         ELSE newcc
       ESAC;


OUTPUT (CASE inst
        OF cmpa¦cmpb¦cpx: input1,
          tpa          newcc
        ELSE             output
      ESAC,  newcc)
END
```

#Decision Logic#

```
FN BRANCH = (mnemonic:inst,address.addrbuf pc,data:cc) -> address:
CASE inst
   OF bra:  addrbuf,
     bcc  CASE TRUEBIT(cc,da/16r01)
            OF l: addrbuf
            ELSE pc
          ESAC,
     bcs: CASE TRUEBIT(cc,da/16r01)
            OF h  addrbuf
            ELSE pc
          ESAC,
     beq  CASE TRUEBIT(cc,da/16r04)
            OF h  addrbuf
            ELSE pc
          ESAC,
     bge  CASE (TRUEBIT(cc,da/16r08),TRUEBIT(cc,da/16r02))
            OF (h,h)|(l,l). addrbuf
            ELSE pc
          ESAC,
     bgt  CASE (TRUEBIT(cc,da/16r04),TRUEBIT(cc,da/16r08),TRUEBIT(cc,da/16r02))
            OF (l,h,h)|(l,l,l)  addrbuf
            ELSE pc
          ESAC,
     bhi  CASE (TRUEBIT(cc,da/16r04),TRUEBIT(cc,da/16r01))
            OF (l,l)  addrbuf
            ELSE pc
          ESAC,
     ble  CASE (TRUEBIT(cc,da/16r04),TRUEBIT(cc,da/16r08),TRUEBIT(cc,da/16r02))
            OF (l,h,h)|(l,l,l)  pc
            ELSE addrbuf
          ESAC,
     bls  CASE (TRUEBIT(cc,da/16r04),TRUEBIT(cc,da/16r01))
            OF (l,l)  pc
            ELSE addrbuf
          ESAC,
     blt  CASE (TRUEBIT(cc,da/16r08),TRUEBIT(cc,da/16r02))
            OF (h,l)|(l,h)  addrbuf
            ELSE pc
          ESAC,
     bmi  CASE TRUEBIT(cc,da/16r08)
            OF h  addrbuf
            ELSE pc
          ESAC,
     bne  CASE TRUEBIT(cc,da/16r04)
            OF l  addrbuf
            ELSE pc
          ESAC,
```

```
        bvc· CASE TRUEBIT(cc,da/16r02)
                OF l: addrbuf
                ELSE pc
             ESAC,
        bvs· CASE TRUEBIT(cc,da/16r02)
                OF h. addrbuf
                ELSE pc
             ESAC,
        bpl: CASE TRUEBIT(cc,da/16r08)
                OF l: addrbuf
                ELSE pc
             ESAC
ESAC.
```

```
#Instruction Decoder#

FN ADMODE = (halfbyte input) -> addrmode
CASE input
   OF hb/16r0¦hb/16r1¦hb/16r3¦hb/16r4¦hb/16r5 implied,
     hb/16r2: relative,
     hb/16r6¦hb/16ra¦hb/16re index,
     hb/16r7¦hb/16rb¦hb/16rf. extnd,
     hb/16r8¦hb/16rc: immed,
     hb/16r9¦hb/16rd direct
ESAC

FN MNEMONIC = (halfbyte: input, halfbyte: mode) -> (mnemonic)
CASE input

   OF hb/16r0: CASE mode
            OF hb/16r1        sba,
              hb/16r2.        bra,
              hb/16r3         tsx,
              hb/16r4         nega,
              hb/16r5         negb,
              hb/(16r6 16r7)  neg,
              hb/(16r8. 16rb) suba,
              hb/(16rc 16rf)  subb
         ESAC,

     hb/16r1: CASE mode
            OF hb/16r1        cba,
              hb/16r3:        ins,
              hb/(16r8 16rb). cmpa,
              hb/(16rc 16rf)  cmpb
         ESAC,

     hb/16r2 CASE mode
            OF hb/16r0        nop,
              hb/16r2         bhi,
              hb/16r3         pula,
              hb/(16r8 16rb)  sbca,
              hb/(16rc 16rf)  sbcb
         ESAC,

     hb/16r3 CASE mode
            OF hb/16r2        bls,
              hb/16r3         pulb,
              hb/16r4         coma,
              hb/16r5         comb,
              hb/(16r6 16r7)  com
         ESAC,
```

```
hb/16r4. CASE mode
        OF hb/16r2          bcc,
           hb/16r3          des,
           hb/16r4          lsra,
           hb/16r5          lsrb,
           hb/(16r6..16r7)  lsr,
           hb/(16r8..16rb)  anda,
           hb/(16rc..16rf)  andb
      ESAC,

hb/16r5 CASE mode
        OF hb/16r2·         bcs,
           hb/16r3·         txs,
           hb/(16r8..16rb). bita,
           hb/(16rc..16rf)  bitb
      ESAC,

hb/16r6 CASE mode
        OF hb/16r0          tap,
           hb/16r1          tab,
           hb/16r2·         bne,
           hb/16r3          psha,
           hb/16r4          rora,
           hb/16r5          rorb,
           hb/(16r6 16r7)   ror,
           hb/(16r8 16rb)   ldaa,
           hb/(16rc 16rf)   ldab
      ESAC,

hb/16r7 CASE mode
        OF hb/16r0          tpa,
           hb/16r1          tba,
           hb/16r2          beq,
           hb/16r3          pshb,
           hb/16r4          asra,
           hb/16r5          asrb,
           hb/(16r6 16r7)   asr,
           hb/(16r9 16rb)   staa,
           hb/(16rd 16rf)   stab
      ESAC,

hb/16r8 CASE mode
        OF hb/16r0          inx,
           hb/16r2          bvc,
           hb/16r4          asla,
           hb/16r5          aslb,
           hb/(16r6 16r7)   asl,
           hb/(16r8 16rb)   eora,
           hb/(16rc 16rf)   eorb
      ESAC,
```

```
hb/16r9. CASE mode
         OF hb/16r0:        dex,
           hb/16r1          daa,
           hb/16r2:         bvs,
           hb/16r3:         rts,
           hb/16r4.         rola,
           hb/16r5:         rolb,
           hb/(16r6 .16r7): rol,
           hb/(16r8 .16rb): adca,
           hb/(16rc..16rf)· adcb
       ESAC,

hb/16ra  CASE mode
         OF hb/16r0         clv,
           hb/16r2:         bpl,
           hb/16r4:         deca,
           hb/16r5·         decb,
           hb/(16r6. 16r7)  dec,
           hb/(16r8 .16rb)  oraa,
           hb/(16rc 16rf)   orab
       ESAC,

hb/16rb  CASE mode
         OF hb/16r0         sev,
           hb/16r1          aba,
           hb/16r2          bmi,
           hb/16r3·         rti,
           hb/(16r8..16rb)  adda,
           hb/(16rc 16rf)   addb
       ESAC,

hb/16rc  CASE mode
         OF hb/16r0         clc,
           hb/16r2          bge,
           hb/16r4          inca,
           hb/16r5          incb,
           hb/(16r6 16r7)   inc,
           hb/(16r8 16rb)   cpx
       ESAC,

hb/16rd  CASE mode
         OF hb/16r0         sec,
           hb/16r2          blt,
           hb/16r4          tsta,
           hb/16r5          tstb,
           hb/(16r6 16r7)   tst,
           hb/16r8          bsr,
           hb/(16ra .16rb)  jsr
       ESAC,
```

```
hb/16re  CASE mode
            OF hb/16r0·      cli,
              hb/16r2·       bgt,
              hb/16r3:       wai,
              hb/(16r6..16r7). jmp,
              hb/(16r8..16rb). lds,
              hb/(16rc. 16rf)  ldx
           ESAC,

hb/16rf. CASE mode
            OF hb/16r0.      sei,
              hb/16r2·       ble,
              hb/16r3        swi,
              hb/16r4.       clra,
              hb/16r5        clrb,
              hb/(16r6..16r7)· clr,
              hb/(16r9 .16rb)· sts,
              hb/(16rd 16rf)  stx
              ESAC
ESAC
```

```
#Counting and Type-Swapping Functions#
FN COUNTSET = (count: a b) -> count:ARITH a+b

FN HOWLONG = (mnemonic function, addrmode mode) -> count.
COUNTSET (CASE function
      OF jmp  co/0,
      staa¦stab¦clra¦clrb¦coma¦comb¦nega¦negb¦deca¦decb¦inca¦incb¦rola¦rolb¦
      rora¦rorb¦asla¦aslb¦asra¦asrb¦lsra¦lsrb¦tsta¦tstb¦cpx¦ldx¦lds¦tba¦tab
      ¦aba¦cba¦daa¦nop¦clc¦cli¦clv¦sec¦sei¦sev¦tap¦tpa  co/2,
      clr¦com¦neg¦dec¦inc¦rol¦ror¦asl¦asr¦lsr¦tst  co/3,
      psha¦pshb¦pula¦pulb¦dex¦des¦inx¦ins¦txs¦tsx¦stx  co/4,
      bsr¦rts  co/5,
      jsr: CASE mode OF extnd  co/6 ELSE co/4 ESAC,
      wai  co/9,
      rti  co/10,
      swi¦nmi¦irq.  co/12
      ELSE co/1
   ESAC,
   CASE mode
      OF immed co/1,
      direct co/2,
      index co/4,
      extnd¦relative co/3
      ELSE co/0
   ESAC)


FN COUNTDOWN = (count  input) -> count ARITH input-1


FN COMPARECOUNT = (count  input1 input2) -> count
      ARITH IF (input1-input2) > 0 THEN input1 - input2 ELSE 0 FI


FN INCADDR = (address input) -> address ARITH IF input = 65535 THEN 0 ELSE input + 1 FI


FN HINYBBLE = (data  input) -> halfbyte ARITH (input IAND 16rf0) SR 4


FN LONYBBLE = (data  input) -> halfbyte ARITH (input IAND 16r0f)


FN MAKEADDRESS = (data  input1 input2) -> address ARITH 256*input1 + input2


FN MAKEDATA = (halfbyte  input1 input2) -> data ARITH 16*input1 + input2


FN HIBYTE = (address  input) -> data ARITH (input IAND 16rff00) SR 8

FN LOBYTE = (address  input) -> data ARITH (input IAND 16r00ff)
```

```
#Main Microprocessor Control Logic#
FN UP6800 = (data input,flag reset birq bnmi halt) -> (data,address,flag)
BEGIN SEQ
PVAR a ::= ?data,
     b ::= ?data,
     x ::= ?address,
    sp : = ?address,
    pc ::= ?address,
    cc . = ?data,
    writeenable.:=?flag,
    addrbus :. = ?address,
    addrbuf ::= ?address,
    databus ·· = ?data,
    intercount ::= ?count,
    admode · = ?addrmode,
    inst ::= ?mnemonic;


CASE halt
   OF h
#----------reset circuit----------#

(  CASE reset
       OF l (pc,admode,inst,intercount,cc)
         · = (ad/16rfffe,extnd,jmp,co/4,da/16rd0)
   ESAC,


#----------housekeeping----------#
   databus  = input,
   writeenable  = l,

   intercount . = COUNTDOWN intercount,


#----------hardware interrupt----------#
CASE intercount
   OF co/0  (CASE bnmi
              OF l  (inst = nmi, intercount = co/12)
          ESAC,
          CASE (birq,TRUEBIT (cc,da/16))
             OF (l,l)  (inst = irq, intercount = co/12)
          ESAC),
     co/1· CASE inst
            OF wai (CASE bnmi
                      OF l. (inst = nmi; intercount = co/4)
                   ESAC,
                   CASE (birq,TRUEBIT (cc,da/16))
                      OF (l,l)  (inst = irq; intercount = co/4)
                   ESAC)
          ESAC
ESAC,
```

```
#------------------treat databus as next instruction-------------------------#

  (inst,admode) := CASE intercount
                   OF co/0:(MNEMONIC (LONYBBLE databus ,HINYBBLE
                                 input),ADMODE HINYBBLE databus)
                   ELSE (inst,admode)
                 ESAC;

  (intercount,pc) := CASE intercount
                     OF co/0: (HOWLONG(inst,admode),INCADDR addrbus)
                     ELSE (intercount,pc)
                   ESAC;

#-----------------------------MAIN PROCEDURE-------------------------------#

CASE COMPARECOUNT (intercount, HOWLONG (inst, implied))
  OF co/4
      CASE admode
        OF index:(addrbus = pc;pc = INCADDR addrbus)
      ESAC,
    co/3
      CASE admode
        OF extnd (addrbus = pc, pc = INCADDR INCADDR addrbus),
          index addrbuf = x,
          relative (addrbus = pc, pc = INCADDR addrbus)
      ESAC,
    co/2
      CASE admode
        OF direct (addrbus = pc, pc = INCADDR addrbus),
          index addrbuf =
          MAKEADDRESS(HIBYTE x,(ALU(adda,LOBYTE x,databus, da'0))[1]),
          extnd (addrbuf,addrbus) =
                (MAKEADDRESS (databus,da/0),INCADDR addrbus),
          relative addrbuf =
            MAKEADDRESS(da'0,(ALU(adda,databus,LOBYTE pc, da/0))[1])
      ESAC,
    co/1
      CASE admode
        OF immed (addrbus = pc, pc = INCADDR addrbus),
          direct addrbus = MAKEADDRESS (da'16r0,databus),
          index (addrbuf =
              MAKEADDRESS ((ALU (adca, HIBYTE x,
                            CASE TRUEBIT (databus,da'16r80)
                              OF h da/16rff
                              ELSE da/16r00
                            ESAC,
                      (ALU (adda,databus,LOBYTE x, da/0))[2]))[1],
                LOBYTE addrbuf),addrbus = addrbuf),
          extnd addrbus = MAKEADDRESS (HIBYTE addrbuf, databus),
```

```
        relative· addrbuf · =
            MAKEADDRESS ((ALU (adca, HIBYTE pc,
                          CASE TRUEBIT (databus,da/16r80)
                             OF h. da/16rff
                             ELSE da/16r00
                          ESAC,
                    (ALU (adda,databus,LOBYTE pc,da/0))[2]))[1],
                     LOBYTE addrbuf)
  ESAC
co/0 CASE COMPARECOUNT (HOWLONG (inst, implied),intercount)
      OF co/0
      CASE
        inst
        OF adda¦adca¦anda¦bita¦clra¦cmpa¦coma¦deca¦eora¦ldaa¦tpa¦
        inca¦nega¦oraa¦rola¦rora¦asla¦asra¦lsra¦sbca¦daa¦suba¦tsta:
           (a, cc, addrbus) =
             BEGIN
               LET acc1 = ALU(inst, a, databus, cc)
               OUTPUT (acc1[1],acc1[2],pc)
             END,
        staa (databus, cc ,writeenable) =
             BEGIN
               LET acc2 = ALU(inst, databus, a, cc)
               OUTPUT (acc2[1],acc2[2],h)
             END,
        addb¦adcb¦andb¦bitb¦clrb¦cmpb¦comb¦decb¦eorb¦ldab¦
        incb¦negb¦orab¦rolb¦rorb¦aslb¦asrb¦lsrb¦sbcb¦subb¦tstb
           (b, cc, addrbus) =
             BEGIN
               LET bcc1 = ALU(inst, b, databus, cc)
               OUTPUT (bcc1[1],bcc1[2],pc)
             END,
        stab (databus, cc ,writeenable) =
             BEGIN
               LET bcc2 = ALU(inst, databus, b, cc)
               OUTPUT (bcc2[1],bcc2[2],h)
             END,
        aba¦cba¦tba (a,cc) =
             BEGIN
               LET acc3 = ALU (inst, a, b, cc)
               OUTPUT (acc3[1],acc3[2])
             END,
        tab. (b,cc) =
             BEGIN
               LET acc4 = ALU (inst, b, a, cc)
               OUTPUT (acc4[1],acc4[2])
             END,
```

```
tst|clr|com|neg|dec|inc|rol|ror|asl|asr|lsr:
   (databus, cc, writeenable) :=
     BEGIN
       LET dbcc1 = ALU(inst, a, databus, cc).
       OUTPUT (dbcc1[1],dbcc1[2],h)
     END,
clc|cli|clv|sec|sei|sev|tap.
                    cc := (ALU(inst,a,databus,cc))[2],
bra|bcc|bcs|beq|bge|bgt|bhi|ble|bls|blt|bmi|bne|bvc|bvs|
bpl. addrbus := BRANCH (inst,addrbuf,pc,cc),
cpx.
  (addrbus := INCADDR addrbus,
   pc := INCADDR pc;
   cc := (ALU (cpx, HIBYTE x, databus, cc))[2]),
ldx
   (addrbus := INCADDR addrbus,
   pc  = CASE admode
          OF immed: INCADDR pc
          ELSE pc
        ESAC,
   (x,cc) =
      BEGIN
        LET xcc = ALU(inst,da/0,databus,cc)
        OUTPUT (MAKEADDRESS(xcc[1], da/0),xcc[2])
      END),
inx|dex
  (x,cc) =
      BEGIN
        LET xcc1 = ALU(inst,LOBYTE x,databus,cc)
        OUTPUT (MAKEADDRESS(HIBYTE x, xcc1[1]),xcc1[2])
      END,
ins|des:
  (sp,cc) . =
      BEGIN
        LET spcc1 = ALU(inst,LOBYTE sp,databus,cc)
        OUTPUT(MAKEADDRESS(HIBYTE sp, spcc1[1]),spcc1[2])
      END,
lds
   (addrbus  = INCADDR addrbus,
   pc := INCADDR pc,
   (sp,cc) =
      BEGIN
        LET spcc2 = ALU(inst,da/0,databus,cc)
        OUTPUT (MAKEADDRESS(spcc2[1],da/0),spcc2[2])
      END),
stx.((databus, cc)  =
      BEGIN
        LET  xcc4 = ALU(stx, databus,HIBYTE x ,cc)
        OUTPUT (xcc4[1],xcc4[2])
      END, writeenable . = h),
```

```
sts:((databus, cc) ·=
      BEGIN
        LET  spcc3 = ALU(sts, databus,HIBYTE sp ,cc)
        OUTPUT (spcc3[1],spcc3[2])
      END, writeenable .= h),
tsx
  (x,cc) :=
      BEGIN
        LET xcc6 = ALU(ins, LOBYTE sp,databus,cc).
        OUTPUT(MAKEADDRESS(HIBYTE sp, xcc6[1]),xcc6[2])
      END,
txs
  (sp,cc) ·=
      BEGIN
        LET xcc7 = ALU(des, LOBYTE x,databus,cc).
        OUTPUT(MAKEADDRESS(HIBYTE x, xcc7[1]),xcc7[2])
      END,
psha  (databus,addrbus,writeenable) .= (a,sp,h),
pshb: (databus,addrbus,writeenable) ·= (b,sp,h),
pula¦pulb· (sp,cc) :=
      BEGIN
        LET spcc4 = ALU(ins,LOBYTE sp,databus,cc)
        OUTPUT(MAKEADDRESS(HIBYTE sp, spcc4[1]),spcc4[2])
      END,
jsr¦bsr·(addrbuf =pc;
     pc =addrbus, addrbus =sp, databus =
     LOBYTE (addrbuf), writeenable = h,
     (sp,cc)  =
      BEGIN
        LET spcc5 = ALU(des,LOBYTE sp,databus,cc)
        OUTPUT(MAKEADDRESS(HIBYTE sp, spcc5[1]),spcc5[2])
      END),
swi¦irq¦nmi¦wai
     (addrbuf =pc, addrbus =sp, databus =
     LOBYTE (addrbuf), writeenable = h,
     (sp,cc)  =
      BEGIN
       LET spcc42 = ALU(des,LOBYTE sp,databus,cc)
       OUTPUT(MAKEADDRESS(HIBYTE sp, spcc42[1]),spcc42[2])
      END),
rts¦rti: (addrbus.= sp, (sp,cc)  =
      BEGIN
       LET spcc6 = ALU(ins,LOBYTE sp,databus cc).
       OUTPUT(MAKEADDRESS(HIBYTE sp,spcc6[1]),spcc6[2])
      END)
ESAC,
```

```
  co/1
CASE inst
    OF stx|sts:addrbus := INCADDR addrbus,
    clra|clrb|coma|comb|nega|negb|deca|decb|inca|incb|rola|
    rolb|rora|rorb|asla|aslb|asra|asrb|lsra|lsrb|tsta|tstb|aba
    |daa|tba|tab|cba|clc|cli|clv|sec|sei|sev|tap|tpa|nop:
    addrbus := pc,
    cpx|ldx:
    (addrbus := pc;
    (x,cc) :=
        BEGIN
          LET xcc2 = ALU(inst,LOBYTE x,databus,cc).
          OUTPUT (MAKEADDRESS(HIBYTE x, xcc2[1]),xcc2[2])
        END),
      lds
    (addrbus := pc,
    (sp,cc) =
        BEGIN
          LET spcc7 = ALU(inst,LOBYTE sp,databus,cc).
          OUTPUT(MAKEADDRESS(HIBYTE sp, spcc7[1]),spcc7[2])
        END),
    psha|pshb (sp,cc) =
        BEGIN
          LET spcc8 = ALU(des,LOBYTE sp,databus,cc)
          OUTPUT(MAKEADDRESS(HIBYTE sp, spcc8[1]),spcc8[2])
        END,
    pula|pulb
    ((sp,cc) =
        BEGIN
          LET spcc9 = ALU(ins,HIBYTE sp,LOBYTE sp,cc).
          OUTPUT (MAKEADDRESS(spcc9[1],LOBYTE sp),spcc9[2])
        END, addrbus := sp),
    dex|inx
    (x,cc)' =
        BEGIN
          LET xcc3 = ALU(inst,HIBYTE x,LOBYTE x,cc)
          OUTPUT (MAKEADDRESS(xcc3[1],LOBYTE x),xcc3[2])
        END,
    des|ins
    (sp,cc) :=
        BEGIN
          LET spcc10 = ALU(inst,HIBYTE sp,LOBYTE sp,cc).
          OUTPUT(MAKEADDRESS(spcc10[1],LOBYTE sp),spcc10[2])
        END,
    tsx
    (x,cc)' :=
        BEGIN
          LET xcc8 = ALU(ins,HIBYTE sp, LOBYTE x,cc)'
          OUTPUT(MAKEADDRESS(xcc8[1],LOBYTE x),xcc8[2])
        END,
```

```
  txs
  (sp, cc) : =
       BEGIN
         LET xcc9 = ALU(des,HIBYTE x, LOBYTE sp,cc).
         OUTPUT(MAKEADDRESS(xcc9[1],LOBYTE sp),xcc9[2])
       END,
  staa¦stab  addrbus . = pc,
  jsr¦bsr¦swi¦irq¦nmi¦wai· ((sp,cc) : =
       BEGIN
         LET spcc11 = ALU(des,HIBYTE sp,LOBYTE sp,cc).
         OUTPUT(MAKEADDRESS(spcc11[1],LOBYTE sp),spcc11[2])
       END;
       addrbus · = sp; databus: = HIBYTE addrbuf;
       writeenable : = h;
      (sp,cc) : =
       BEGIN
         LET spcc12 = ALU(des,LOBYTE sp,databus,cc)
         OUTPUT(MAKEADDRESS(HIBYTE sp,spcc12[1]),spcc12[2])
       END),
  rts¦rti: ((sp,cc)  =
       BEGIN
         LET spcc13 = ALU(ins,HIBYTE sp,LOBYTE sp,cc)
         OUTPUT(MAKEADDRESS(spcc13[1],LOBYTE sp),spcc13[2])
       END, addrbus  = sp, (sp,cc)  =
       BEGIN
         LET spcc14 = ALU(ins,LOBYTE sp,databus,cc).
         OUTPUT(MAKEADDRESS(HIBYTE sp,spcc14[1]),spcc14[2])
       END)
ESAC,
```

```
  co/2·
CASE inst
    OF stx((databus, cc) : =
        BEGIN
          LET xcc5 = ALU(stx, databus,LOBYTE x ,cc).
          OUTPUT (xcc5[1],xcc5[2])
        END; writeenable · = h),
      tst|clr|com|neg|dec|inc|rol|ror|asl|asr|lsr
      addrbus : = pc,
      psha|pshb: (sp,cc) : =
        BEGIN
          LET spcc15 = ALU(des,HIBYTE sp,LOBYTE sp,cc)
          OUTPUT(MAKEADDRESS(spcc15[1],LOBYTE sp),spcc15[2])
        END,
      pula: a· = databus,
      pulb: b· = databus,
      jsr|bsr:  (sp,cc) : =
        BEGIN
          LET spcc16 = ALU(des,HIBYTE sp,LOBYTE sp,cc)
          OUTPUT(MAKEADDRESS(spcc16[1],LOBYTE sp),spcc16[2])
        END,
      rts  ((sp,cc)  =
        BEGIN
          LET spcc17 = ALU(ins,HIBYTE sp,LOBYTE sp,cc)
          OUTPUT(MAKEADDRESS(spcc17[1],LOBYTE sp),spcc17[2])
        END; addrbus : = sp;
        pc : = MAKEADDRESS(databus,da/0)),
      swi|irq|nmi|wai
        ((sp,cc) · =
        BEGIN
          LET spcc18 = ALU(des,HIBYTE sp,LOBYTE sp,cc)
          OUTPUT(MAKEADDRESS(spcc18[1],LOBYTE sp),spcc18[2])
        END,
        addrbus  = sp, databus = LOBYTE x,
        writeenable  = h,
        (sp,cc)  =
        BEGIN
          LET spcc19 = ALU(des,LOBYTE sp,databus,cc)
          OUTPUT(MAKEADDRESS(HIBYTE sp,spcc19[1]),spcc19[2])
        END),
      rti  ((sp,cc)  =
        BEGIN
          LET spcc20 = ALU(ins,HIBYTE sp,LOBYTE sp,cc)
          OUTPUT(MAKEADDRESS(spcc20[1],LOBYTE sp),spcc20[2])
        END, addrbus  = sp,
        cc = databus,
        (sp,cc) · =
        BEGIN
          LET spcc21 = ALU(ins,LOBYTE sp,databus,cc)
          OUTPUT(MAKEADDRESS(HIBYTE sp,spcc21[1]),spcc21[2])
        END)
ESAC,
```

```
  co/3:
 CASE inst
   OF rts|stx|psha|pula|pshb|pulb|dex|des|inx|ins|txs|tsx:
     addrbus := pc,
     rts: pc := MAKEADDRESS(HIBYTE pc,databus),
     jsr|bsr: addrbus := pc,
     swi|irq|nmi|wai ((sp,cc) :=
         BEGIN
          LET spcc22 = ALU(des,HIBYTE sp,LOBYTE sp,cc).
          OUTPUT(MAKEADDRESS(spcc22[1],LOBYTE sp),spcc22[2])
         END, addrbus := sp; databus. = HIBYTE x;writeenable := h; (sp,cc) :=
         BEGIN
          LET spcc23 = ALU(des,LOBYTE sp,databus,cc).
          OUTPUT(MAKEADDRESS(HIBYTE sp,spcc23[1]),spcc23[2])
         END),
     rti. ((sp,cc) . =
         BEGIN
          LET spcc24 = ALU(ins,HIBYTE sp,LOBYTE sp,cc).
          OUTPUT(MAKEADDRESS(spcc24[1],LOBYTE sp),spcc24[2])
         END; addrbus . = sp; b = databus, (sp,cc) : =
         BEGIN
          LET spcc25 = ALU(ins,LOBYTE sp,databus,cc)
          OUTPUT(MAKEADDRESS(HIBYTE sp,spcc25[1]),spcc25[2])
         END)
 ESAC,
   co/4
 CASE inst
   OF rts: addrbus. = pc,
     swi|irq|nmi|wai ((sp,cc) =
         BEGIN
          LET spcc26 = ALU(des,HIBYTE sp,LOBYTE sp,cc)
          OUTPUT(MAKEADDRESS(spcc26[1],LOBYTE sp),spcc26[2])
         END,
         addrbus = sp, databus = a,
         writeenable = h,
        (sp,cc) =
         BEGIN
          LET spcc27 = ALU(des,LOBYTE sp,databus,cc)
          OUTPUT(MAKEADDRESS(HIBYTE sp,spcc27[1]),spcc27[2])
         END),
     rti ((sp,cc) =
         BEGIN
          LET spcc28 = ALU(ins,HIBYTE sp,LOBYTE sp,cc)
          OUTPUT(MAKEADDRESS(spcc28[1],LOBYTE sp),spcc28[2])
         END, addrbus = sp,
         a = databus,
         (sp,cc) =
         BEGIN
          LET spcc29 = ALU(ins,LOBYTE sp,databus,cc)
          OUTPUT(MAKEADDRESS(HIBYTE sp,spcc29[1]),spcc29[2])
         END)
 ESAC,
```

```
 co/5.
CASE inst
  OF swi|irq|nmi|wai
       ((sp,cc) :=
       BEGIN
        LET spcc30 = ALU(des,HIBYTE sp,LOBYTE sp,cc).
        OUTPUT(MAKEADDRESS(spcc30[1],LOBYTE sp),spcc30[2])
       END,
       addrbus := sp; databus := b; writeenable := h,
      (sp,cc) :=
       BEGIN
        LET spcc31 = ALU(des,LOBYTE sp,databus,cc)
        OUTPUT(MAKEADDRESS(HIBYTE sp,spcc31[1]),spcc31[2])
       END),
    rti: ((sp,cc) :=
       BEGIN
        LET spcc32 = ALU(ins,HIBYTE sp,LOBYTE sp,cc)
        OUTPUT(MAKEADDRESS(spcc32[1],LOBYTE sp),spcc32[2])
       END; addrbus := sp, x := MAKEADDRESS(databus,da/0);
       (sp,cc) . =
       BEGIN
        LET spcc33 = ALU(ins,LOBYTE sp,databus,cc).
        OUTPUT(MAKEADDRESS(HIBYTE sp,spcc33[1]),spcc33[2])
       END)
 ESAC,
  co/6
CASE inst
  OF swi|irq|nmi|wai
       ((sp,cc) =
       BEGIN
        LET spcc34 = ALU(des,HIBYTE sp,LOBYTE sp,cc)
        OUTPUT(MAKEADDRESS(spcc34[1],LOBYTE sp),spcc34[2])
       END,
       addrbus = sp, databus = cc,
       writeenable = h,
      (sp,cc) . =
       BEGIN
        LET spcc35 = ALU(des,LOBYTE sp,databus,cc)
        OUTPUT(MAKEADDRESS(HIBYTE sp,spcc35[1]),spcc35[2])
       END),
    rti ((sp,cc) =
       BEGIN
        LET spcc36 = ALU(ins,HIBYTE sp,LOBYTE sp,cc).
        OUTPUT(MAKEADDRESS(spcc36[1],LOBYTE sp),spcc36[2])
       END, addrbus = sp,
       x = MAKEADDRESS(HIBYTE x,databus),
       (sp,cc) =
       BEGIN
        LET spcc37 = ALU(ins,LOBYTE sp,databus,cc)
        OUTPUT(MAKEADDRESS(HIBYTE sp,spcc37[1]),spcc37[2])
       END)
 ESAC,
```

```
            co/7:
          CASE inst
            OF swi|irq|nmi|wai.
                 ((sp,cc) ·=
                 BEGIN
                   LET spcc38 = ALU(des,HIBYTE sp,LOBYTE sp,cc).
                   OUTPUT(MAKEADDRESS(spcc38[1],LOBYTE sp),spcc38[2])
                 END; cc:= (ALU(sei,da/0,da/0,cc))[2]),
               rti: ((sp,cc) :=
                 BEGIN
                   LET spcc39 = ALU(ins,HIBYTE sp,LOBYTE sp,cc).
                   OUTPUT(MAKEADDRESS(spcc39[1],LOBYTE sp),spcc39[2])
                 END, addrbus := sp,
                 pc = MAKEADDRESS (databus,da/0))
          ESAC,
            co/8
          CASE inst
            OF swi addrbus = ad/16rfffa,
              irq addrbus = ad/16rfff8,
              nmi addrbus:= ad/16rfffc,
              rti pc = MAKEADDRESS (HIBYTE pc,databus),
              wai intercount = co/2
          ESAC,
            co/9·
          CASE inst
            OF swi:(pc.= MAKEADDRESS(databus,da/0), addrbus.= ad/16rfffb),
              irq:(pc = MAKEADDRESS(databus,da/0), addrbus· = ad/16rfff9),
              nmi (pc = MAKEADDRESS(databus,da/0), addrbus = ad/16rfffd),
              rti addrbus = pc
          ESAC,
            co/10
          CASE inst
            OF swi|irq|nmi pc = MAKEADDRESS(HIBYTE pc,databus)
          ESAC,
            co/11
          addrbus = pc
        ESAC
ESAC)
ESAC,

OUTPUT (databus,addrbus,writeenable)
END
```

```
#External Memory and Test Harness#

FN MAKERAM = (data,address,address,flag) -> data RAM (da/16r0).

FN DEL = (data) -> data: DELAY (da/16r0,1)

FN HARNESS = (flag·reset birq bnmı, data  hexf000 hexf001 hexf002) -> [3]data·
BEGIN
MAKE UP6800: micproc,
    DEL: delay olatch1 olatch2 olatch3,
    MAKERAM: ram


JOIN reset -> micproc[2],
    CASE micproc[2]
        OF ad/16rf000· hexf000,
           ad/16rf001: hexf001,
           ad/16rf002: hexf002
       ELSE ram
    ESAC -> delay,
    delay -> micproc[1],
    (micproc[1],micproc[2],micproc[2],micproc[2])-> ram,
    birq -> micproc[3],   #ırq#
    bnmı -> micproc[4],   #nmı#
    h -> micproc[5],      #halt#
    CASE micproc[2]
        OF ad/16re000. micproc[1]
        ELSE olatch1
    ESAC -> olatch1,
    CASE micproc[2]
        OF ad/16re001  micproc[1]
        ELSE olatch2
    ESAC -> olatch2,
    CASE micproc[2]
        OF ad/16re002  micproc[1]
        ELSE olatch3
    ESAC -> olatch3


OUTPUT (olatch1,olatch2,olatch3)
END
```

Appendix B: Programs

The following programs all work on my ELLA description of the 6800 microprocessor. They are held on disk by the ELLA group at DRA Malvern To run any program the programs initialise and monitor must be stored as [6800]microinit eli and [6800]monitor.eli since all the other programs call these

Note: i) co is a simulator command for comment I have left them in for completeness

ii) These programs have been modified slightly from when they were originally written because of changes to the reset circuitry. They run on the model described above

iii) If you wish to see the microprocessor operating the following node names are useful to monitor:

micproc pc	program counter
micproc.admode	addressing mode
micproc.inst	instruction mnemonic
micproc.databus	value on the data bus
micproc addrbus	value on the address bus
micproc.a	accumulator a
micproc.b	accumulator b
micproc.x	index register
micproc sp	stack pointer
micproc cc	condition code register

Initialise

```
co This program performs a reset by setting parameter 1 low for 1 timestep, clears the a, b and x, registers,
co sets the stack pointer to &7fff, clears the condition codes register and jumps to location &000a
simulatefn HARNESS
cp 1 h h da/0 da/0 da/0
iram ram [16r8000] da/16r4f, co   reset   clra
iram ram [16r8001] da/16r5f, co           clrb
iram ram [16r8002] da/16r8e, co           lds  #&7fff
iram ram [16r8003] da/16r7f,
iram ram [16r8004] da/16rff,
iram ram [16r8005] da/16rce, co           ldx  #&0000
iram ram [16r8006] da/16r00,
iram ram [16r8007] da/16r00,
iram ram [16r8008] da/16r06, co           tap
iram ram [16r8009] da/16r7e, co           jmp  &000a
iram ram [16r800a] da/16r00
iram ram [16r800b] da/16r0a
iram ram [16rfffe] da/16r80, co   define address to jump to on reset
iram ram [16rffff] da/16r00
ti +1
cp [1] h
```

Monitor

```
co This program allows the online changing of a program by performing an nmi
co (make input [3] low for 1 cycle with inputs [4. 6] holding the appropriate
co values). Input [4] controls the function as follows:
co da/0  advance counter and display with contents of that location
co da/1  change contents of location to value of input [5]
co da/2  load counter with contents of input [5..6]
co da/3  execute your program starting at location defined by counter
co Thus to call the monitor type (from the simulator) cp [3 .6] l da/2 da/<address_n> da/<address_s>,ti +7,
co cp [3] h, ti +10.

in [6800]microinit
iram ram [16r9000] da/16rb6, co  .monitor   ldaa &f000
iram ram [16r9001] da/16rf0
iram ram [16r9002] da/16r00
iram ram [16r9003] da/16r84, co             anda #&0f
iram ram [16r9004] da/16r0f
iram ram [16r9005] da/16r81, co             cmpa #&03
iram ram [16r9006] da/16r03
iram ram [16r9007] da/16r23, co             bls  .valid
iram ram [16r9008] da/16r01
iram ram [16r9009] da/16r3b, co             rti
iram ram [16r900a] da/16r8e, co  valid      lds  #&7fff
iram ram [16r900b] da/16r7f
iram ram [16r900c] da/16rff
iram ram [16r900d] da/16r81, co             cmpa #&01
iram ram [16r900e] da/16r01
iram ram [16r900f] da/16r26, co             bne  nochange
iram ram [16r9010] da/16r03
iram ram [16r9011] da/16rbd, co             jsr  .change
iram ram [16r9012] da/16r91
iram ram [16r9013] da/16r00
iram ram [16r9014] da/16r81, co  nochange   cmpa #&01
iram ram [16r9015] da/16r01
iram ram [16r9016] da/16r22, co             bhi  .noadv
iram ram [16r9017] da/16r01
iram ram [16r9018] da/16r08, co             inx
iram ram [16r9019] da/16r81, co  noadv      cmpa #&02
iram ram [16r901a] da/16r02
iram ram [16r901b] da/16r26, co             bne  oldx
iram ram [16r901c] da/16r03
iram ram [16r901d] da/16rfe, co             ldx  &f001
iram ram [16r901e] da/16r00
iram ram [16r901f] da/16r01
iram ram [16r9020] da/16r81, co  .oldx      cmpa #&03
iram ram [16r9021] da/16r03
iram ram [16r9022] da/16r26, co             bne  print
iram ram [16r9023] da/16r02
iram ram [16r9024] da/16r6e, co             jmp  &00, x
iram ram [16r9025] da/16r00
iram ram [16r9026] da/16re6, co  .print     ldab &00, x
iram ram [16r9027] da/16r00
iram ram [16r9028] da/16rf7, co             stab &e002
iram ram [16r9029] da/16re0
iram ram [16r902a] da/16r02
iram ram [16r902b] da/16rff, co             stx  &e000
```

```
iram ram [16r902c] da/16re0
iram ram [16r902d] da/16r00
iram ram [16r902e] da/16r3e, co            wai
iram ram [16r9100] da/16rf6, co  .change   ldab &f001
iram ram [16r9101] da/16rf0
iram ram [16r9102] da/16r01
iram ram [16r9103] da/16re7, co            stab &00,x
iram ram [16r9104] da/16r00
iram ram [16r9105] da/16r39, co            rts
iram ram [16rfffc] da/16r90, co   define nmi address
iram ram [16rfffd] da/16r00
```

Testprog1

co This is a very simple program which prints out the numbers 1,4,7, by adding 3 to the previous number co etc.

```
in [6800]monitor
iram ram [10] da/16r86, co   .start  ldaa #&01
iram ram [11] da/16r01
iram ram [12] da/16r8b, co           adda #&03
iram ram [13] da/16r03
iram ram [14] da/16rb7, co           staa &e000
iram ram [15] da/16re0
iram ram [16] da/16r00
iram ram [17] da/16r7e, co           jmp  &000c
iram ram [18] da/16r00
iram ram [19] da/16r0c
tabulated
mc
```

Testprog2

```
co This program is the first in a series of Fibonacci sequence programs  It
co uses only the a register and only direct, extended and relative addressing
co It was written to run on the early model which only had six instructions
in [6800]monitor
iram ram [10] da/16r86, co  .start ldaa #&00
iram ram [11] da/16r00
iram ram [12] da/16r97, co         staa &00
iram ram [13] da/16r00
iram ram [14] da/16r97, co  .loop  staa &03
iram ram [15] da/16r03
iram ram [16] da/16r86, co         ldaa #&01
iram ram [17] da/16r01
iram ram [18] da/16r97, co         staa #&01
iram ram [19] da/16r01
iram ram [20] da/16r96, co         ldaa &01
iram ram [21] da/16r00
iram ram [22] da/16r9b, co         adda
iram ram [23] da/16r01
iram ram [24] da/16r97, co         staa &02
iram ram [25] da/16r02
iram ram [26] da/16rb7, co         staa &e000
iram ram [27] da/16re0
iram ram [28] da/16r00
iram ram [29] da/16r96, co         ldaa &01
iram ram [30] da/16r01
iram ram [31] da/16r97, co         staa &00
iram ram [32] da/16r00
iram ram [33] da/16r96, co         ldaa &02
iram  am [34] da/16r02
iram ram [35] da/16r97, co         staa &01
iram ram [36] da/16r01
iram ram [37] da/16r96, co         ldaa &03
iram ram [38] da/16r03
iram ram [39] da/16r8b, co         adda #&01
iram ram [40] da/16r01
iram ram [41] da/16r97, co         staa &03
iram ram [42] da/16r03
iram ram [43] da/16r80, co         suba #&0a
iram ram [44] da/16r0a
iram ram [45] da/16r27, co         bne   skip
iram ram [46] da/16r03
iram ram [47] da/16r7e, co         jmp   loop
iram ram [48] da/16r00
iram ram [49] da/16r14
iram ram [50] da/16r7e, co         jmp   start
iram ram [51] da/16r00
iram ram [52] da/16r0a
tabulated
mc
```

Testprog3

co This Fibonnacci program uses indexed addressing

```
in [6800]monitor
iram ram [10] da/16r86, co  start  ldaa #&00
iram ram [11] da/16r00,
iram ram [12] da/16r97, co         staa &00
iram ram [13] da/16r00,
iram ram [14] da/16r97, co         staa &03
iram ram [15] da/16r03,
iram ram [16] da/16r86, co         ldaa #&01
iram ram [17] da/16r01,
iram ram [18] da/16r97, co         staa &01
iram ram [19] da/16r01,
iram ram [20] da/16rc6, co         ldab #&0a
iram ram [21] da/16r0a,
iram ram [22] da/16r96, co .main   ldaa &00
iram ram [23] da/16r00,
iram ram [24] da/16r9b, co         adda &01
iram ram [25] da/16r01,
iram ram [26] da/16r97, co         staa &02
iram ram [27] da/16r02,
iram ram [28] da/16rb7, co         staa &e000
iram ram [29] da/16re0,
iram ram [30] da/16r00,
iram ram [31] da/16rce, co         ldx  #&0001
iram ram [32] da/16r00,
iram ram [33] da/16r01,
iram ram [34] da/16ra6, co .loop   ldaa &00,x
iram ram [35] da/16r00,
iram ram [36] da/16r09, co         dex
iram ram [37] da/16ra7, co         staa &00,x
iram ram [38] da/16r00,
iram ram [39] da/16r08, co         inx
iram ram [40] da/16r08, co         inx
iram ram [41] da/16r8c, co         cpx  &0003
iram ram [42] da/16r00,
iram ram [43] da/16r03,
iram ram [44] da/16r26, co         bne  loop
iram ram [45] da/16rf4,
iram ram [46] da/16r5a, co         decb
iram ram [47] da/16r26, co         bne  .main
iram ram [48] da/16re5,
iram ram [49] da/16r7e, co         jmp  start
iram ram [50] da/16r00,
iram ram [51] da/16r0a,
tabulated
mc
```

Testprog4

```
co This program tests the x register and stack pointer, it places the ten Fibonnacci
co numbers generated on the stack and outputs them in reverse order.
in [6800]monitor
initialiseram ram [10] da/16r86, co  .start  ldaa #&00
initialiseram ram [11] da/16r00,
initialiseram ram [12] da/16r97, co          staa &00
initialiseram ram [13] da/16r00,
initialiseram ram [14] da/16r97, co          staa &03
initialiseram ram [15] da/16r03,
initialiseram ram [16] da/16r8e, co          lds  #&0403
initialiseram ram [17] da/16r04,
initialiseram ram [18] da/16r03,
initialiseram ram [19] da/16r86, co          ldaa #&01
initialiseram ram [20] da/16r01,
initialiseram ram [21] da/16r97, co          staa &01
initialiseram ram [22] da/16r01,
initialiseram ram [23] da/15rc6, co          ldab &0a
initialiseram ram [24] da/16r0a,
initialiseram ram [25] da/16r96, co  .main   ldaa &00
initialiseram ram [26] da/16r00,
initialiseram ram [27] da/16r9b, co          adda &01
initialiseram ram [28] da/16r01,
initialiseram ram [29] da/16r97, co          staa &02
initialiseram ram [30] da/16r02,
initialiseram ram [31] da/16r36, co          psha
initialiseram ram [32] da/16rce, co          ldx  #&0001
initialiseram ram [33] da/16r00,
initialiseram ram [34] da/16r01,
initialiseram ram [35] da/16ra6, co  loop    ldaa &00,x
initialiseram ram [36] da/16r00,
initialiseram ram [37] da/16r09, co          dex
initialiseram ram [38] da/16ra7, co          staa &00,x
initialiseram ram [39] da/16r00,
initialiseram ram [40] da/16r08, co          inx
initialiseram ram [41] da/16r08, co          inx
initialiseram ram [42] da/16r8c, co          cpx  &0003
initialiseram ram [43] da/16r00,
initialiseram ram [44] da/16r03,
initialiseram ram [45] da/16r26, co          bne  loop
initialiseram ram [46] da/16rf4,
initialiseram ram [47] da/16r5a, co          decb
initialiseram ram [48] da/16r26, co          bne  main
initialiseram ram [49] da/16re7,
initialiseram ram [50] da/16rce, co          ldx  #&000a
initialiseram ram [51] da/16r00,
initialiseram ram [52] da/16r0a,
initialiseram ram [53] da/16r32, co  loop2   pula
initialiseram ram [54] da/16rb7, co          staa &e000
initialiseram ram [55] da/16re0,
initialiseram ram [56] da/16r00,
initialiseram ram [57] da/16r09, co          dex
initialiseram ram [58] da/16r26, co          bne  loop2
initialiseram ram [59] da/16rf9,
initialiseram ram [60] da/16r7e, co          jmp  start
```

initialiseram ram [61] da/16r00,
initialiseram ram [62] da/16r0a,
tabulated
mc

Testprog5

co This is another Fibonacci program which tests the jsr instruction

```
in [6800]monitor
initialiseram ram [10] da/16r86, co  .start  ldaa #&00
initialiseram ram [11] da/16r00,
initialiseram ram [12] da/16r97, co          staa &00
initialiseram ram [13] da/16r00,
initialiseram ram [14] da/16r97, co          staa &03
initialiseram ram [15] da/16r03,
initialiseram ram [16] da/16r8e, co          lds  #&0401
initialiseram ram [17] da/16r04,
initialiseram ram [18] da/16r01,
initialiseram ram [19] da/16r86, co          ldaa #&01
initialiseram ram [20] da/16r01,
initialiseram ram [21] da/16r97, co          staa &01
initialiseram ram [22] da/16r01,
initialiseram ram [23] da/16rc6, co          ldab &0a
initialiseram ram [24] da/16r0a,
initialiseram ram [25] da/16r96, co  main    ldaa &00
initialiseram ram [26] da/16r00,
initialiseram ram [27] da/16r9b, co          adda &01
initialiseram ram [28] da/16r01,
initialiseram ram [29] da/16r97, co          staa &02
initialiseram ram [30] da/16r02,
initialiseram ram [31] da/16rb7, co          staa &e000
initialiseram ram [32] da/16re0,
initialiseram ram [33] da/16r00,
initialiseram ram [34] da/16r36, co          psha
initialiseram ram [35] da/16rbd, co          jsr  swap
initialiseram ram [36] da/16r00,
initialiseram ram [37] da/16r35,
initialiseram ram [38] da/16r5a, co          decb
initialiseram ram [39] da/16r26, co          bne  main
initialiseram ram [40] da/16rf0,
initialiseram ram [41] da/16rce, co          ldx  #&000a
initialiseram ram [42] da/16r00,
initialiseram ram [43] da/16r0a,
initialiseram ram [44] da/16r32, co  loop2   pula
initialiseram ram [45] da/16r97, co          staa &02
initialiseram ram [46] da/16r02,
initialiseram ram [47] da/16r09, co          dex
initialiseram ram [48] da/16r26, co          bne  loop2
initialiseram ram [49] da/16rfa,
initialiseram ram [50] da/16r7e, co          jmp  start
initialiseram ram [51] da/16r00
initialiseram ram [52] da/16r0a,
initialiseram ram [53] da/16rce, co  .swap   ldx  #&0000
initialiseram ram [54] da/16r00,
initialiseram ram [55] da/16r00,
initialiseram ram [56] da/16ra6, co  loop    ldaa &01, x
initialiseram ram [57] da/16r01,
initialiseram ram [58] da/16ra7, co          staa &00, x
initialiseram ram [59] da/16r00,
initialiseram ram [60] da/16r08, co          inx
```

```
initialiseram ram [61] da/16r8c, co      cpx  &0002
initialiseram ram [62] da/16r00,
initialiseram ram [63] da/16r02,
initialiseram ram [64] da/16r26, co      bne  .loop
initialiseram ram [65] da/16rf6,
initialiseram ram [66] da/16r39, co      rts
tabulated
mc
```

Testprog6

```
co This is yet another Fibonacci program, which uses a software interrupt (swi)
co instead of the jsr in the last program (testprog5)

in [6800]monitor
initialiseram ram [10] da/16r86, co  .start  ldaa #&00
initialiseram ram [11] da/16r00,
initialiseram ram [12] da/16r97, co          staa &00
initialiseram ram [13] da/16r00,
initialiseram ram [14] da/16r97, co          staa &03
initialiseram ram [15] da/16r03,
initialiseram ram [16] da/16r8e, co          lds  #&0401
initialiseram ram [17] da/16r04,
initialiseram ram [18] da/16r01,
initialiseram ram [19] da/16r86, co          ldaa #&01
initialiseram ram [20] da/16r01,
initialiseram ram [21] da/16r97, co          staa &01
initialiseram ram [22] da/16r01,
initialiseram ram [23] da/16rc6, co          ldab &0a
initialiseram ram [24] da/16r0a,
initialiseram ram [25] da/16r96, co  .main   ldaa &00
initialiseram ram [26] da/16r00,
initialiseram ram [27] da/16r9b, co          adda &01
initialiseram ram [28] da/16r01,
initialiseram ram [29] da/16r97, co          staa &02
initialiseram ram [30] da/16r02,
initialiseram ram [31] da/16rb7, co          staa &e000
initialiseram ram [32] da/16re0,
initialiseram ram [33] da/16r00,
initialiseram ram [34] da/16r36, co          psha
initialiseram ram [35] da/16r3f, co          swi
initialiseram ram [36] da/16r5a, co          decb
initialiseram ram [37] da/16r26, co          bne  .main
initialiseram ram [38] da/16rf2,
initialiseram ram [39] da/16rce, co          ldx  #&000a
initialiseram ram [40] da/16r00,
initialiseram ram [41] da/16r0a,
initialiseram ram [42] da/16r32, co  loop2   pula
initialiseram ram [43] da/16r97, co          staa &02
initialiseram ram [44] da/16r02,
initialiseram ram [45] da/16r09, co          dex
initialiseram ram [46] da/16r26, co          bne  loop2
initialiseram ram [47] da/16rfa,
initialiseram ram [48] da/16r7e, co          jmp  start
initialiseram ram [49] da/16r00,
initialiseram ram [50] da/16r0a,
initialiseram ram [51] da/16rce, co  swap    ldx  #&0000
initialiseram ram [52] da/16r00,
initialiseram ram [53] da/16r00,
initialiseram ram [54] da/16ra6, co  loop    ldaa &01, x
initialiseram ram [55] da/16r01,
initialiseram ram [56] da/16ra7, co          staa &00, x
initialiseram ram [57] da/16r00,
initialiseram ram [58] da/16r08, co          inx
initialiseram ram [59] da/16r8c, co          cpx  &0002
```

```
initialiseram ram [60] da/16r00,
initialiseram ram [61] da/16r02,
initialiseram ram [62] da/16r26, co          bne  .loop
initialiseram ram [63] da/16rf6,
initialiseram ram [64] da/16r3b, co          rti
initialiseram ram [16rfffa] da/16r00, co     define swi address
initialiseram ram [16rfffb] da/16r33, co     .swap
tabulated
mc
```

Testprog7

co This is an 19 byte program which generates the first 11 Fibonacci numbers It operates by clearing one co location on the stack to zero, then loading the a register with &01 and x with &000b. Accumulator b is co then pulled from the stack, then a is pushed to it. Register a is added to b with the result in a. x is co decremented. If x is not zero then b is pulled from the stack, a is pushed to it etc. Otherwise the program co waits for an interrupt. It is short because it was written to be efficient rather than to test specific co instructions. However Testprog2 performs an identical function and I believe is optimised for the first co 6 instructions implemented (see The Motorola 6800 Model. initial design) but is twice as long This co demonstrates that more complex operations are useful on a microprocessor

```
in [6800]monitor
iram ram [16r000a] da/16r34, co    start   des
iram ram [16r000b] da/16r30, co            tsx
iram ram [16r000c] da/16r6f, co            clr  &00 ,x
iram ram [16r000d] da/16r00
iram ram [16r000e] da/16rce, co            ldx  #&000b
iram ram [16r000f] da/16r00
iram ram [16r0010] da/16r0b
iram ram [16r0011] da/16r86, co            ldaa #&01
iram ram [16r0012] da/16r01
iram ram [16r0013] da/16r33, co    .loop    pulb
iram ram [16r0014] da/16r36, co            psha
iram ram [16r0015] da/16r1b, co            aba
iram ram [16r0016] da/16rb7, co            staa &e000
iram ram [16r0017] da/16re0
iram ram [16r0018] da/16r00
iram ram [16r0019] da/16r09, co            dex
iram ram  [16r001a] da/16r26, co           bne  .loop
iram ram [16r001b] da/16rf7
iram ram [16r001c] da/16r3e, co            wai
tabulated
mc
```

Appendix C: The Motorola 6800 Instruction Set

Provided here is a brief list of the 6800 assembler mnemonics and their functions It is not intended to provide programming information, which may be obtained from The Motorola Data Sheet (see Bibliography)

Accumulator and Memory Instructions:

adda	: Add.	a = a + m
addb	: Add·	b = b + m
aba	: Add accumulators	a = a + b
adca	· Add with carry:	a = a + m + c
adcb	. Add with carry:	b = b + m + c
anda/andb	. And	a = a AND m / b = b AND m
bita/bitb	Bit test	a AND m / b AND m
clr/clra/clrb	. Clear	m = 0 / a = 0 / b = 0
cmpa/cmpb	Compare	a-m / b-m
cba	· Compare accumulators	a-b
com/coma/comb	· 1's complement.	m = &ff-m / a = &ff-a / b = &ff-b
neg/nega/negb	2's complement (negate)	m = 0-m / a = 0-a / b = 0-b
daa	decimal adjust a	
dec/deca/decb	decrement	m = m-1 / a = a-1 / b = b-1
eora/eorb	exclusive or	a XOR m / b XOR m
inc/inca/incb	. increment	m = m + 1 / a = a + 1 / b = b + 1
ldaa/ldab	load accumulator	a = m / b = m
oraa/orab	· inclusive or	a = a OR m / b = b OR m
psha/pshb	· push a / b to stack (LIFO)	
pula/pulb	pull a / b from stack	
rol/rola/rolb	· rotate left m / a / b	
ror/rora/rorb	· rotate right m / a / b	
asl/asla/aslb	: arithmetic shift left m / a / b	
asr/asra/asrb	· arithmetic shift right m / a / b	
lsr/lsra/lsrb	· logic shift right m / a / b	
staa/stab	store accumulator	m = a / m = b
suba/subb	subtract	a = a-m / b = b-m
sba	subtract accumulators	a = a-b
sbca/sbcb	subtract with carry	a = a-m-c / b = b-m-c
tab/tba	transfer accumulators	b = a / a = b
tst/tsta/tstb	· test, zero or minus	m-0 / a-0 / b-0

Index Register and Stack Manipulation Instructions

cpx	. compare x to m, m + 1	
dex/des	decrement	x = x-1 / sp = sp-1
inx/ins	increment	x = x + 1 / sp = sp + 1
ldx/lds	· load register	$x_h = m$, $x_l = (m + 1)$ / $sp_h = m$, $sp_l = (m + 1)$
stx/sts	store register	$m = x_h$, $(m + 1) = x_l$ / $m = sp_h$, $(m + 1) = sp_l$
txs/tsx	transfer registers	sp = x-1 / x = sp + 1

Jump and Branch Instructions:

bra	branch always
bcc	branch if carry clear
bcs	branch if carry set
beq	branch if equals zero
bge	branch if greater than or equals zero
bgt	branch if greater than zero
bhi	branch if higher
ble	branch if less than or equals zero
bls	branch if lower or same
blt	branch if less than zero
bmi	branch if minus
bne	branch if not equal zero
bvc	branch if overflow clear
bvs	branch if overflow set
bpl	branch if plus
bsr	branch to subroutine
jmp	jump
jsr	jump to subroutine
nop	no operation (takes a small amount of time)
rti	return from interrupt
rts	return from subroutine
swi	software interrupt
wai	wait for interrupt

Dummy Mnemonics Added to Simplify ELLA Program:

irq	interrupt request
nmi	non-maskable interrupt

Condition Code Register Manipulation Operations:

clc	clear carry flag
cli	clear interupt mask
clv	clear overflow flag
sec	set carry flag
sei	set interrupt mask
sev	set overflow
tap	transfer accumulator a into cc register
tpa	transfer cc register into accumulator a

a : accumulator a b : accumulator b m : contents of memory location
(m+1) : contents of next memory location after m x : x register sp : stack pointer
$_{h}$: high byte $_{l}$: low byte

N.B Many Mnemonics can have more than one addressing mode associated with them There are altogether 197 valid mnemonic/adressing mode combinations plus irq and nmi, although there are only 107 such mnemonics listed (plus 2 dummy mnemonics).

Glossary:

Accumulator·	An accumulator is a *register* on which a large number of arithmetic and logical operations may be carried out.
Addressing mode.	The addressing mode tells the microprocessor where to get the data for its next operation Six addressing modes are implemented on the Motorola 6800. These are *immediate, direct, indexed, extended, implied* and *relative*
Assembler code.	Low level language which is made up of *mnemonics* plus other symbols to indicate the *addressing mode*.
Condition codes register:	A *register* which is affected by most arithmetic and logical functions, and also by *interrupts*. On the 6800 it is made up of 6 flags, which indicate when true (bit 5 first, bit 0 last) that: a half carry from bit 3 has occurred, *interrupt requests* are to be ignored, the result was negative, the result was zero, the operation caused a 2's complement overflow and the operation caused a carry.
Direct addressing	The address of the data is given by the value in the next location after the *op-code*, (therefore it is in the range &0000 &00ff).
Extended addressing	The address of the data is in the next two locations after the *op-code* (high byte first)
Immediate addressing	The data comes from the next memory location after the *op-code*
Implied addressing	No data is necessary e.g clra
Indexed addressing	The address of the data is the index register plus an offset which is in the next location after the *op-code*
Interrupt	An interrupt causes the microprocessor to stop what it is doing after finishing the current instruction, and to jump to a location which is defined at a particular location in memory When the routine is finished (with an rti) it jumps back to where it was when the interrupt occurred, with the same values in all the *registers* as before the interrupt This is achieved by dumping all the registers on the *stack* as the interrupt is initiated, and reading them back as the interrupt routine terminates There are three ways of initiating an interrupt on the 6800, a *software interrupt* (swi), and two hardware interrupts
Interrupt request	If the interrupt mask bit in the *condition codes register* is zero, a zero on the interrupt request line causes an *interrupt* to occur.
Mnemonic	A mnemonic is a three or four letter word which defines the operation to be performed by the microprocessor The words are chosen to be user friendly and must be converted into *op-codes* before they can be stored in memory or used by the microprocessor
Non-maskable interrupt	A non-maskable interrupt causes an *interrupt* whatever the state of the *condition codes register*

Op-code:	Hexadecimal representation of an 8-bit number which is an instruction to a microprocessor. An op-code is unique, defining both the *addressing mode* and the *mnemonic*.
Register:	A register is a store for information on the microprocessor A limited number of arithmetic and logical functions are possible such as increment, decrement, or setting particular bits to be true or false.
Relative addressing:	The address to branch to is the start of the next instruction plus an offset which comes from the location after the current *op-code*.
Software interrupt.	A particular *op-code* is used from within the program to call the *interrupt* routine
Stack.	A stack is an area of memory used for storing data, which is accessed by the stack pointer. The stack pointer (sp) always points to the first available unused location When a piece of data is pushed to the stack it is placed at the memory location sp, and sp is then decremented. When a piece of data is pulled from the stack the stack pointer is first incremented, then the data is read The location defined by sp is now free for new data to be pushed, but still contains the old data The stack is therefore a last-in first-out (LIFO) store
Von Neumann microprocessor:	A microprocessor in which the program and data share a common address space

Bibliography:

1 . . . The ELLA User Manual, Computer General ED

2 . . Microprocessor Data, Cambridge University Engineering Dept.

3 . . Motorola Microprocessor, Microcontroller and Peripheral Data Volume 1, Motorola Limited

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
MEMO 4515	AUGUST	1991

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
USING ELLA FOR HIGH LEVEL DESIGN:
MODELLING THE MOTOROLA 6800 MICROPROCESSOR

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
HUGGETT, A R	52

Abstract

This document starts by providing a brief overview of the ELLA language The majority of the document describes the design and development of an abstract high level ELLA model of a Motorola 6800 Microprocessor, including all of the 197 codes which make up the 6800 instruction set

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED